IM3080 Design and Innovation Project
(AY2023/24 Semester 1)
Project Report

Title: CLUTCH, Car Listings and User-Targeted Competitive Holdings
Github: https://github.com/UnknownDelta/Auctionable
Submitted by: Group 5
Supervisor: Mr Chua Hock Chuan

1. Background and Motivation

The automotive industry has witnessed the increasing demand in the use of online platforms to facilitate vehicle sales and transactions. This phenomenon is due to the inconvenience nature of the conventional method of potential and current vehicle owners having to go through a third party to purchase or sell vehicles. Traditional methods of buying and selling vehicles include physical auctions, which can be time consuming and have limited outreach, especially in Singapore. Vehicle auctions require the physical presence of buyers, and the limited space puts a cap on the number of participants. Furthermore, the lack of transparency in the process makes it more challenging for users to access the wide range of vehicles and be actively updated on auction results.

Keeping all these inconveniences and limitations in mind, this project aims to address those challenges by creating a user-friendly mobile application for online vehicle auctions and by streamlining the process of auctioning vehicles.

2. Objective

Our objective is to design a smartphone application which eases the selling and buying of used cars without going through a third-party intermediary such as salespersons. It gives buyers a peace of mind as the cars have gone through vigorous inspections to ensure the seller have not falsified features or aspects of the car. Sellers would be happy for this application as if the buyer would back out from the deal after guaranteeing to buy it, they would pay a cancellation fee as to prevent fraud.

To differentiate ourselves to the existing application available in the market, we have implemented auctions. Auctions add an element of enjoyment as individuals compete for favourable deals. Additionally, it enhances the desirability of the vehicle by showcasing the collective interest through bidding activities.

3. Literature Review

3.1 Carousell

Carousell is an online marketplace platform where users can sell and buy items without relying on intermediary between transactions. The company is valued at $1.1 billion dollars, proving the success of the demands and services of the marketplace. Furthermore, the platform can garner many users due to its simplicity of appearance and operations for customers.

3.2 Carro

Carro is a leading online used car marketplace in Southeast Asia, with over 2 million active users [1]. The company's success lies in its comprehensive ecosystem that encompasses a full suite of car-related services, including car buying, selling, financing, insurance, maintenance, and repair [2]. This integrated approach has resonated with customers, making Carro a trusted and preferred choice for their automotive needs. As a forerunner in the marketplace of used cars, Carro serves as a benchmark and a valuable case study for understanding the factors that contribute to the success of such platforms.

3.3 Auction

According to Oxford Dictionary, “A public sale in which each bidder offers an increase upon the offered by the preceding,”

Online auctions are a popular way to buy and sell items, and the various types of auctions each have their own unique features. Factors such as the auction process and the number of bidders can significantly influence the outcome of an online auction. Most online auctions are timed, as they create a sense of urgency and encourage bidders to make decisions quickly.

Timed auctions also streamline the auction process, providing a clear endpoint for buyers, and promoting efficiency and predictability for all users. The formats used in the auction process to determine the winner are also considered. The following formats of auctions were of interest to our group:

3.3.1 English Auction

An English Auction begins with the auctioneer setting parameters like the starting price and bid increment. Interested buyers bid on the car, incrementally increasing its price. The auction closes at the scheduled end time, and the highest bidder pays the final price.

This type of auction is simple to understand, even for users who are not familiar with auctions. It is fully transparent, and all the bidders can observe the current bid price before placing their own bids.

3.3.2 Penny Auction

Penny auctions are a type of online auction where participants pay a small non-refundable fee for each bid they place. The auction timer resets with each new bid, and the last bidder wins the item at the final bid price. This collective payment structure allows the winner to win the item at a bargain price, while the seller profits from the cumulative bid fees, often exceeding the actual item’s cost.

While penny auctions may appear to offer great deals, they are often criticized for their gambling-like nature and potential for deceptive practices. Due to the non-refundable nature of bid fees, penny auctions can be addictive and lead to excessive spending. Despite these concerns, there are a few exceptions to the gambling-like aspects of penny auctions. For instance, websites like QuiBids, restrict the entry of new bidders after a certain limit is reached, while websites like MadBid implement an 'Earned Discount' feature that converts all unsuccessful bids into shopping vouchers, allowing users to redirect their bidding expenses towards other items. These measures help to reduce the gambling-like nature of penny auctions. Overall, penny auctions are exciting, but users must be fully aware of the risks before bidding.

3.4 Git

Linus Torvalds, the developer of Linus, with others in the Linux development community designed and published Git in 2005. It is a version control application that allows numerous developers to easily work on a project. It aids in the monitoring of changes in source code during software development, allowing engineers to work on various elements of a project concurrently without conflict. In our project, we used git to allow us to work on various branches of the project without interfering with each other's code. Once finished, changes contributed by our team members can be integrated, making the project's development more efficient and faster.

3.5 Jira

Jira, which Atlassian created in 2002, has matured into a popular project management platform that allows teams to organize projects, detect issues, and automate workflows. We utilized Jira in our project to lay down the exact tasks that were required to construct our mobile application in our project and planned and allocated a specified amount of time to finish them. Jira also assisted us in determining if we were on track and ensuring that we were not behind schedule.

3.6 GitHub

GitHub - originated in 2008 by Tom Preston-Werner, Chris Wanstrath, and PJ Hyett - was a worldwide frenzy for leveraging version control systems. Git version control is a widely used web-based platform for collaborative software development where teams can efficiently organise code, track modifications and seamlessly coordinate their coding projects, facilitating a more collaborative approach to software development. To work on this project simultaneously, our group created a GitHub repository for increased convenience as it allows us to reduce instances of duplicating work and can easily revert to previous versions if needed.

3.7 Software Development Life Cycle

Software Development Life Cycle (SDLC) refers to the cost-effectiveness and time efficiency to help design and build for the final project. Each companies have a different number of phases, most ranging between five to seven phases. The seven common phases are planning, analysis, design, implementation, testing, deployment and maintenance. By following the phases, the project will minimize the risk taken while also outputting a high-quality product. As such it is important to follow the SDLC the company had given to achieve a successful product.

3.8 Frontend Development

Frontend Development refers to creating and designing the user interface (UI) and the user experience (UX). A front-end developer utilizes programming languages that caters to the user first hand. Allowing them to interact with the site or app. Frontend gives the user the first impression of the app. If the app has a confusing and unappealing UI, the app would have failed from retaining nor entice the user to use this app. It would also give the user an unprofessional and untrustworthy impression, thus would lower the apps reputation. Having a proper navigation would give an easy and fast experience, thus making the user more inclined to continue using the app.

3.9 Backend Development

Backend Development refers to working server-side software, focuses on the everything not seen by the website. It ensures the website is working smoothly. The Frontend refers the outer look while the Backend refers the inner workings of the app. Backend development requires the servers and databases to help store, fetch and delete listings using Database software maintaining it. Examples being MongoDB, MYSQL, PostgreSQL and Oracle. Backend communicates with the Frontend, as it controls what information is pushed to the Frontend. Using Application Program Interface to help integrate new applications with pre-existing software which help streamline process and not create code from scratch.

4. Design and Implementation

Before we think of the design. We think up what is the functionality we want to include in our app. We brainstorm how our page is navigated to have a seamless experience. We took inspiration from pre-existing apps such as Carousell and Carro. Thus, we took the easily accessibility and user-friendliness of Carousell with the professionalism and sleek design of Carro.

4.1 Problem Statement

Transacting vehicles directly between consumers is a tedious and risky process. Sourcing the right buyer can take a long time, filling in paperwork and conducting thorough condition checks all require professional help. These problems force consumers to traditional car dealership who takes a sizeable cut of the sale as their profits.

4.2 Design Consideration

In crafting the design of our app, careful consideration was given to user experience, with an emphasis on intuitive navigation and a visually appealing interface. The choice of components, such as colour palette, typography, and icons reflect a cohesive design strategy aimed at enhancing usability and creating a seamless interaction for our diverse user base.

4.2.1 Colour Palette

Figure 1: 2 prominent colours in CLUTCH app (in hexadecimal)

Our application focuses on two colours; Blue and Green. More specifically, in terms of hexadecimal colours, '#0077B5' and '#00A859'. The shade of blue often carries a sense of trustworthiness and reliability, making it a popular choice for networking, corporate and business applications. Meanwhile, the shade of green represents lively and positivity, making it suitable for a dynamic and growth-oriented environment. The combination of deep blue and vibrant green creates a balanced and engaging colour palette that can enhance the user experience on our app for selling and auctioning vehicles.

4.2.2 Typography

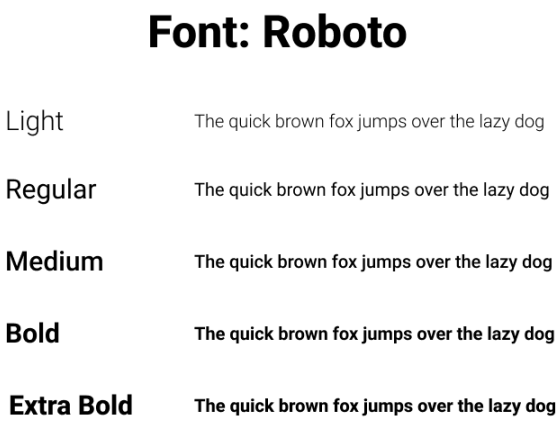


Figure 2: Roboto text font and style

On our app, we implement Roboto font type. Roboto is a clean and modern design text that practises readability on digital screen and ensuring that users can effortlessly navigate through the platform. The font's versatility offers a range of weights and styles, allowing us to establish a clear hierarchy for headings, subheading and body text. The typeface's brand neutrality aligns with our goal of prioritising the vehicle' prominence, keeping the focus on the cars rather than the font. Its availability and support for both Android and IOS further ensures a consistent typographic experience for user accessing our app through different devices, contributing to an overall polished and professional presentation. In essence, the user of Roboto underscores

our commitment to delivering a visually appealing, readable and cohesive typography experience, enhancing the user journey within our innovative vehicle marketplace.

4.2.3 Icons

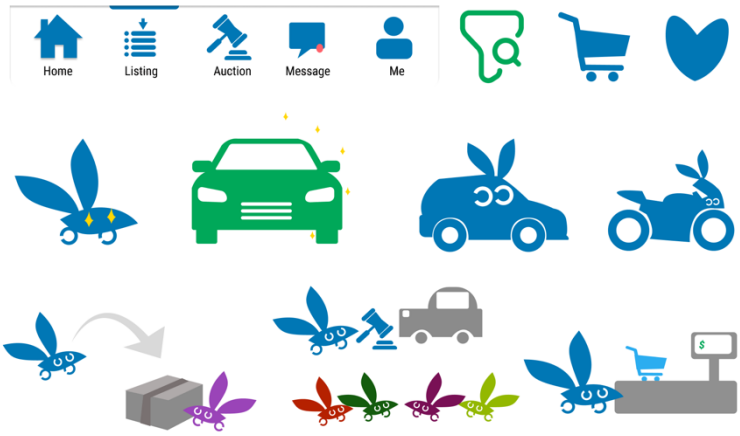


Figure 3: Various icons and logos appear in CLUTCH app

Icons play a crucial role in presentation and enhancing user interaction and navigation. The vector-based icons ensure scalability and adaptability, allowing for seamless appearance on various devices and screen sizes without compromising visual quality. The inherent flexibility of vector graphics ensures that our icons maintain sharpness and clarity, offering a polished and profession appearance. Technicality-wise, the lightweight nature of vector icons enables smooth transitions for the overall user experience and contribute to faster loading time, optimising the application's performance.

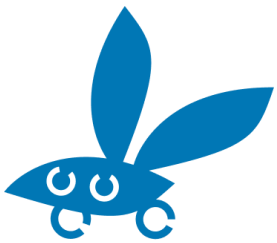


Figure 4: CLUTCH mascot character, a rabbit with wheels.

The mascot character that can be seen throughout our app is an adorable blue rabbit with wheels. The character constitutes a distinctive and visually arresting character, poised to leave a lasting impression in the minds of users. It also symbolises speed and movement, which are both important aspect of an app like ours where we sell and auction vehicles. The vector design of the character makes it easy to use in variety of different contexts such as our app's logo, marketing materials and social media posts. In the local context, rabbit and vehicle are symbols that encapsulate

Singapore's unique blend of harmony, prosperity, innovation and technology. The simplicity of the character creates a strong and positive brand identity for CLUTCH.

4.3 Conceptual Stages

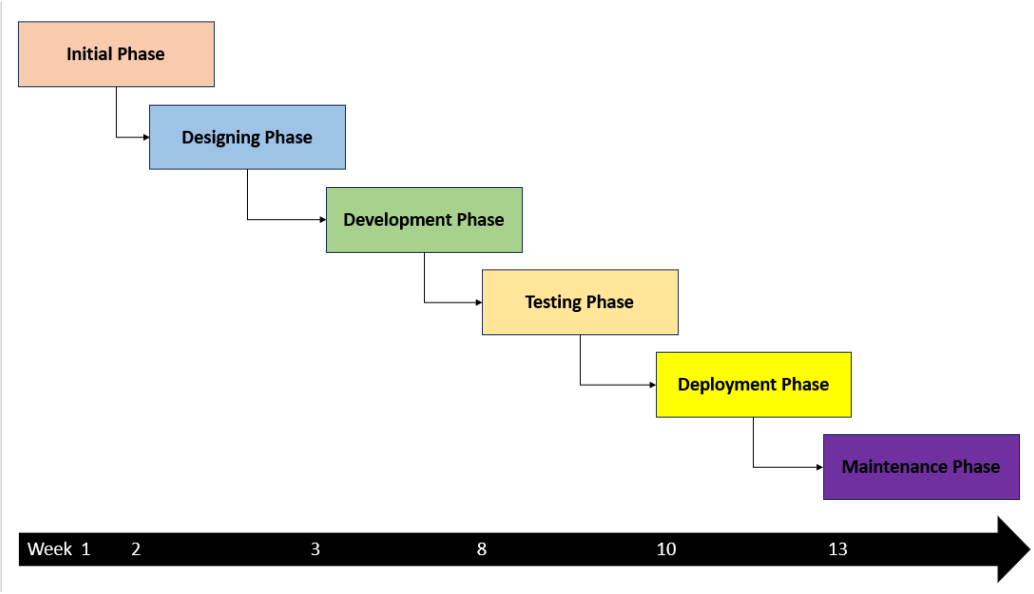


Figure 5: A waterfall model of our project

4.3.1 Initial Stage

The initial phase of our mobile application project involved a thorough analysis of the problem statement, purpose, and target audience. By carefully examining existing real-world problems faced by everyday people, we were able to identify a significant gap in the market. After engaging in a comprehensive group discussion, we decided to develop a marketplace where users could seamlessly sell, buy, and auction vehicles. The team then explored various software options to determine the most suitable tools for the project. This initial phase spanned two weeks.

4.3.2 Designing Phase

This phase focuses on drafting wireframes and solidifying the project purpose. The team had also used this phase to familiarise with the relevant tools, and software to be used in our project:

- **Figma** is used to create and design the wireframe prototypes for our application interfaces as the platform is easier to work with in a simultaneously collaboration environment.

For the app-development, we decided to use the **MERN** stack, which is made up of MongoDB, Express.js, React Native and Node.js.

- **MongoDB** is used because it is the most popular NoSQL. Its ease of access of stored data and is faster compared to using MySQL. It could also able to handle a large dataset, thus making it perfect for our project.

- **React Native** is used to create the app interface, following the design on Figma. We chose to utilise React Native due to its ability to develop cross-platform applications and enable us to view code changes easily.

- **NodeJS** is used because it is efficient and easy to quickly deploy a web application. It has less overhead as compared to Java, thus making it ideal for our objective.

- **Amazon Web Service (AWS)** is used because it is reliable in protecting the data. It helps with running services that might be to intensive client side. AWS is reliable in being secure and mitigation against Distributed Denial of Service (DDoS).

The second phase spanned over five weeks, laying a solid foundation for the subsequence stages.

4.3.3 Development Phase

The third phrase involves constructing the fundamental framework of the application in accordance with the utilised software and the functionalities designated by both frontend and backend teams. This stage serves as an opportunity to assess and determine the practical implementation of features and functions. Beyond the programming aspect, discussion take place to incorporate the refinements and ideas proposed by the design and policy teams. This process ensures the practicality and viability of our project as it progresses to subsequent stages. This phase spanned and overlapped between week 3 to week 8.

4.3.4 Testing and Deployment Phase

The fourth and fifth phase running concurrently as the team integrates their works into a single functional mobile application. Several tweaks and relocation of features are needed to provide better performance of the project. At this stage, the team has successfully operated the several functions:

- Listing, buying, selling and auctioning vehicles
- Optimising the search function
- Model message environment
- Displaying auction activity
- Translating design wireframes into functional pages
- Receiving notifications

The collaborative phase extended from week 8 to week 13, marking the penultimate stage before the project's dateline.

4.3.5 Maintenance Phase

The last phase continues monitoring and troubleshooting the project to address any unforeseen issues that may arise. The goal is to uphold the application's effectiveness and respond promptly to feedback for ongoing improvements on the CLUTCH mobile application. At this stage, the team has delivered project's objective by introducing the resolvent for the problem statement that was voiced out from the initial phase.

4.4 Features

4.4.1 Shop/Listing

After the user have logged in to the app, the user is greeted to the front page of the app, with icons of famous car brands for a quick access of a brand they are interested in. Below the categories of brands, the front page shows popular listing which multiple people are bidding on and a car listing which is highly liked by app users.

After the user pressed see more, it leads to a listing page which shows cars recently posted to the app. At the top of the listing page, there is a filter which allows the users to sort based on the criteria the user have input. From the listing, it shows at a glance, the buying price, the seller profile picture and how long the car have been used.

Clicking the listing page, it brings the product page. It shows the description of the cars written by the seller. The pictures are uploaded by the seller. It features a button to bid in an auction or chat with the seller to either negotiate for a lower price or buy the car as the given price.

4.4.2 Auction

Auctions allow Sellers to swiftly liquidate their vehicles. Sellers can list their vehicle up for auction, while choosing the auction duration, reserve, and buyout price. To

preserve the legitimacy of the auction, a 5% deposit of the reserve price will be collected before the listing can go live.

Auctions allow Buyers to purchase vehicles at the price that they are willing to pay without having to waste time on negotiations. A small deposit will be collected once bids cross the reserve price to ensure the legitimacy of the auction. Once the auction is successful, payment will be done to Clutch, and Clutch will do all the necessary paperwork and condition check before finalizing the sale.

4.5 Diagrams

4.5.1 Use case diagram

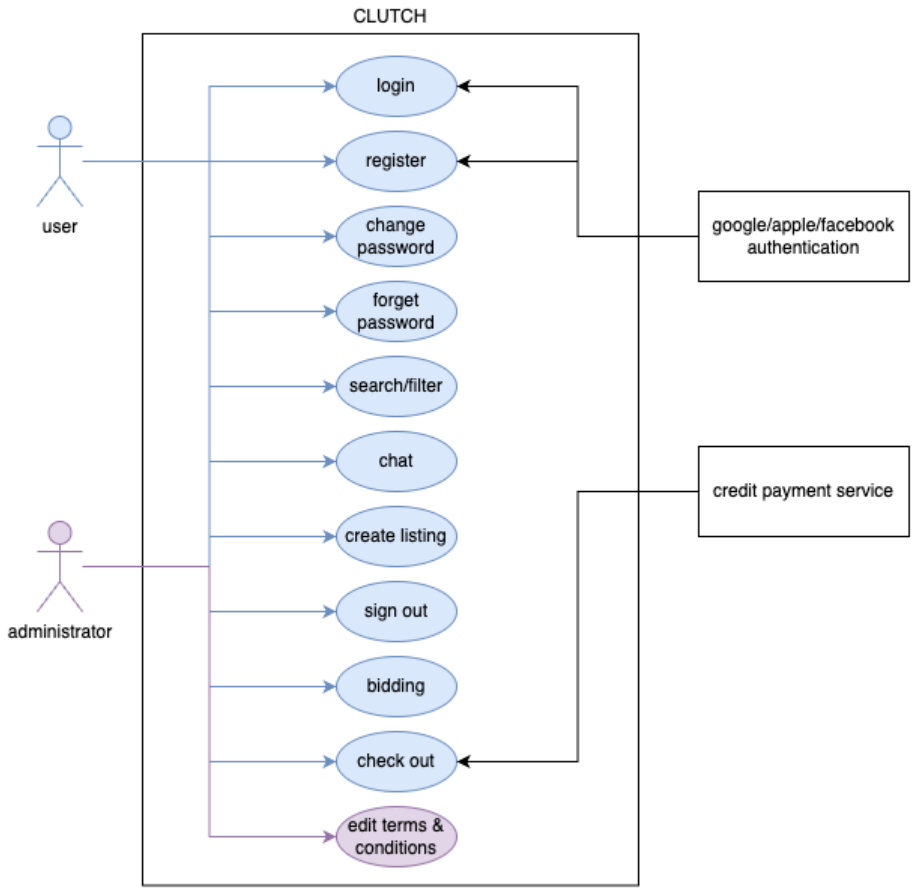


Figure 6. Use cases of CLUTCH

4.5.2 Sequence diagrams

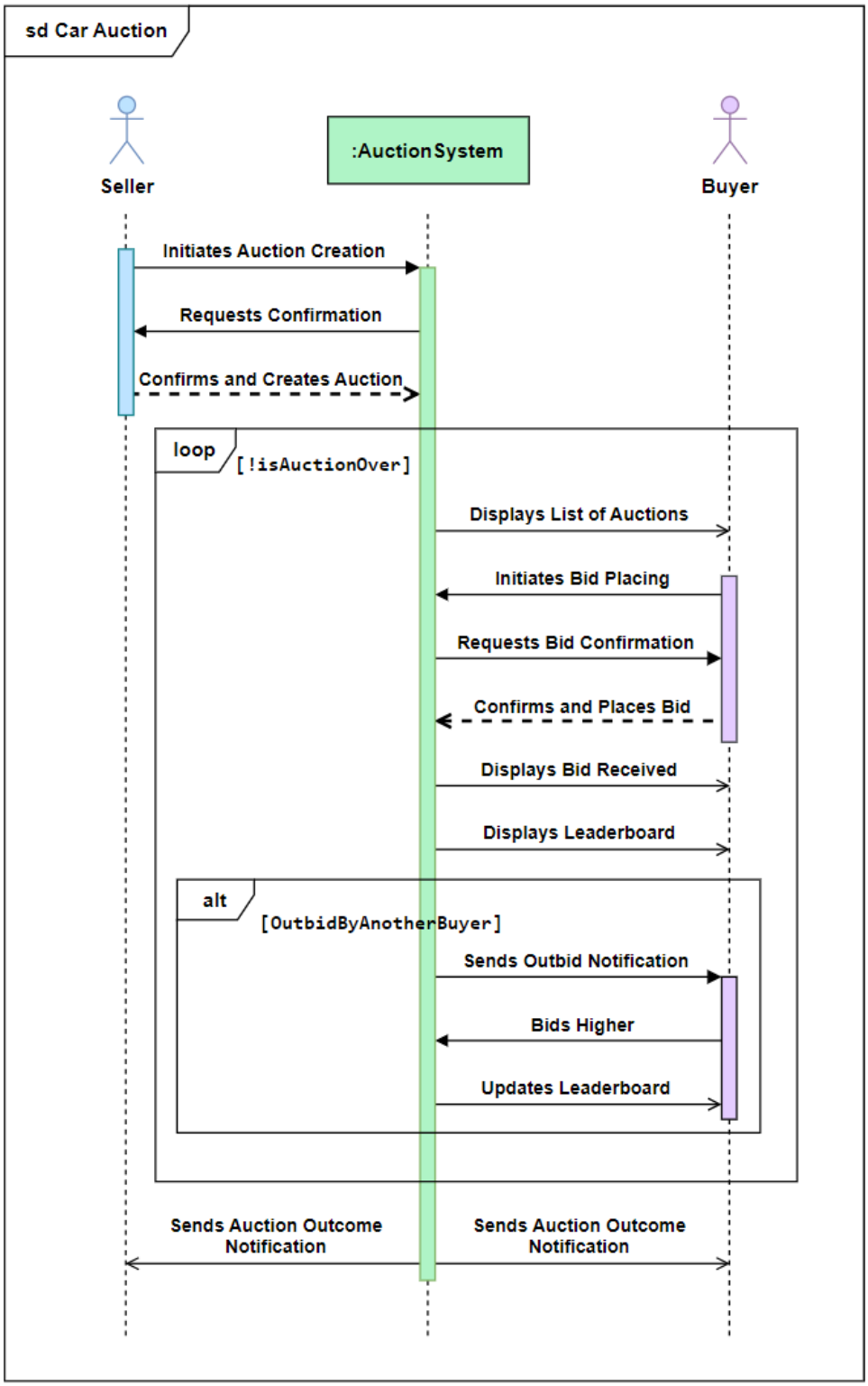


Figure 7. Sequence Diagram for Auctioning Cars

The sequence diagram illustrating the interactions during the process of auctioning cars is shown in Figure 7. The seller initiates the auction by creating a listing. While the auction has not ended, the buyer can place bids, with the option to counterbid in the case of being outbid. The auction system dynamically manages the bids, updating the highest bid and notifying the relevant parties.

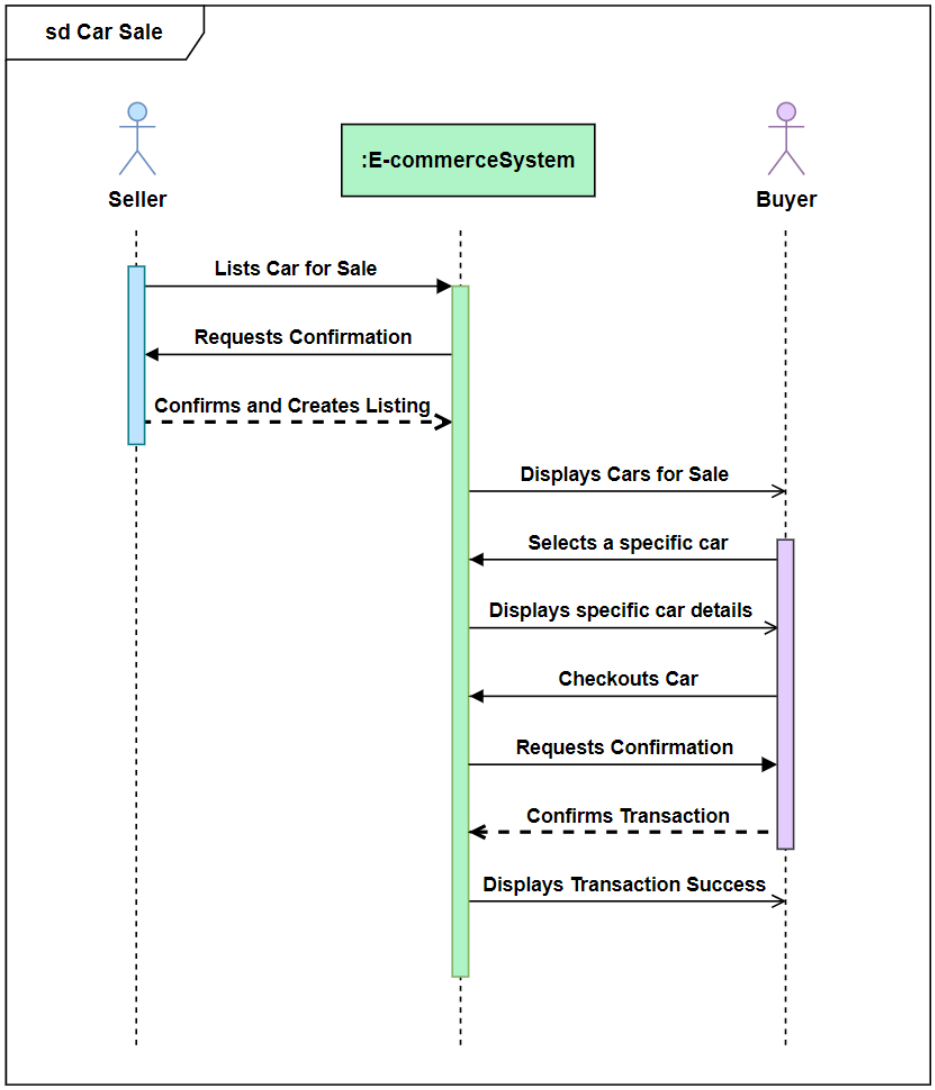


Figure 8. Sequence Diagram for Selling Cars

Figure 8 shows the sequence diagram of the process of selling cars through our platform. In this scenario, buyers can browse and search for cars of interest. Upon selecting, they can proceed to checkout, make payment, and receive confirmation of a successful transaction from the system.

4.6 Figma Wireframe

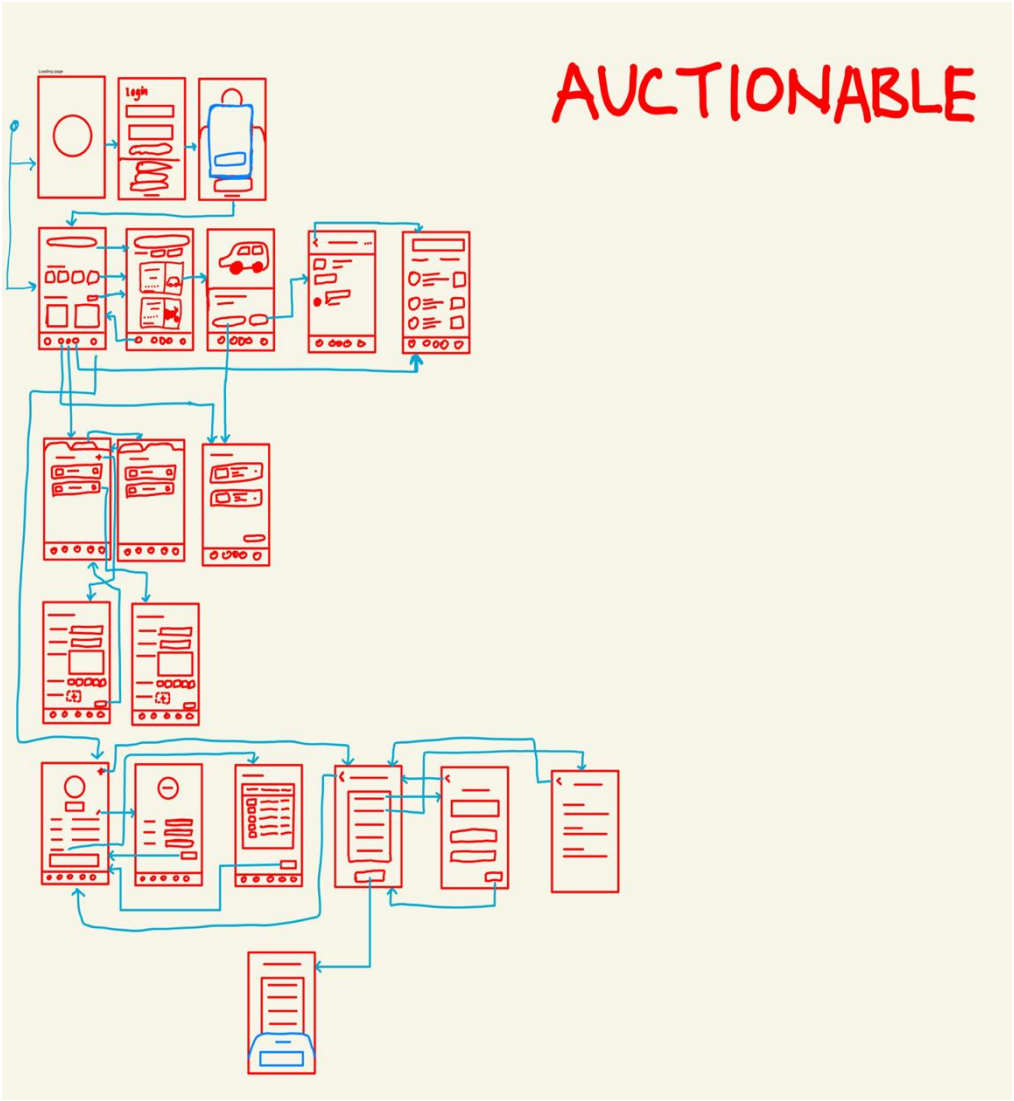


Figure 9: Initial Wireframe of 'AUCTIONABLE', a prototype name prior to naming it CLUTCH

Figure 9 illustrates the blueprint for the user interface. Some of the prominent pages that has been developed into our application such as:

- Login Pages
- Main Pages
- Listing Pages
- Item Pages

The figure below is the refined version of the user interface design that is prominent and implemented into our application:

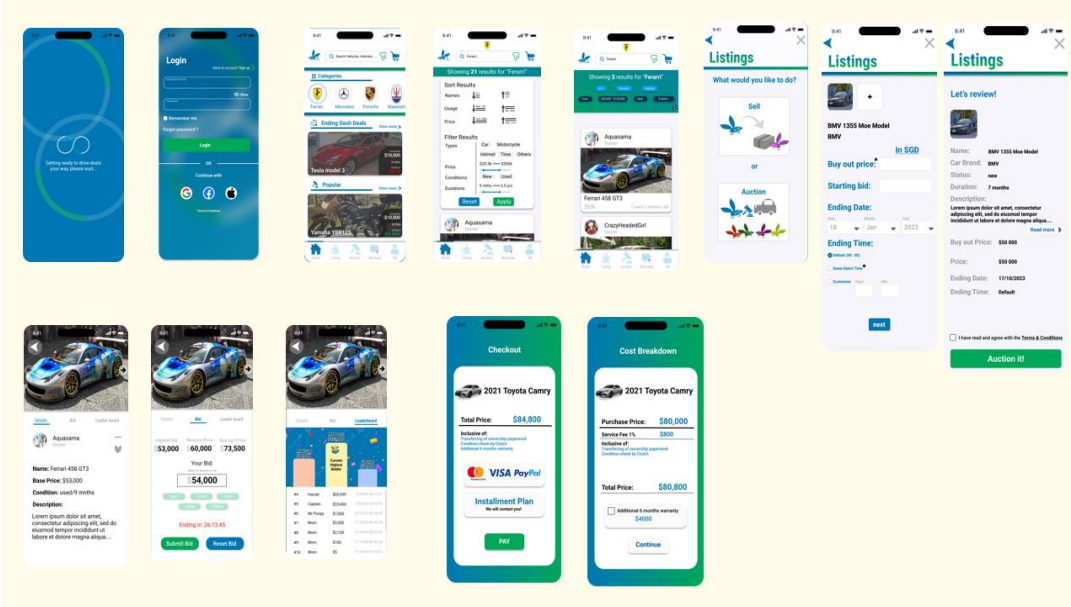


Figure 10: Several prominent and important finalised pages implemented into CLUTCH

4.7 Folder Structure

The backend of our project is organized into three main sections for clarity and efficiency. The first section is dedicated to the Express Framework running on Node.js. in Figure B.01, which forms the server's core, handling HTTP requests and responses necessary for the application's operation. The second section contains the routers and controllers for MongoDB in Figure B.01, which are crucial for database operations, managing the data flow, and executing database queries. The final section focuses on DevOps with AWS Amplify in Figure B.02, which we use for deployment purposes, ensuring our application is consistently and reliably delivered to the users.

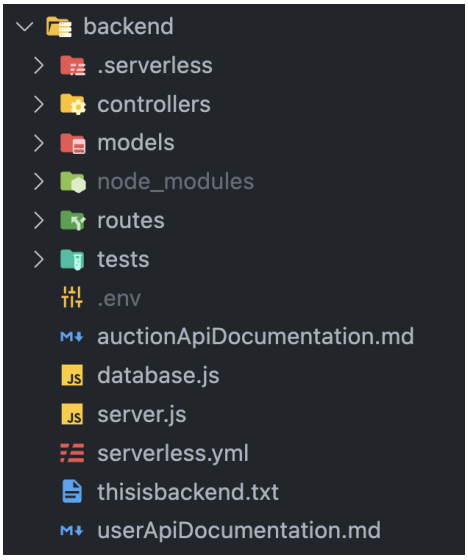


Figure 11: Express.Js and MongoDB

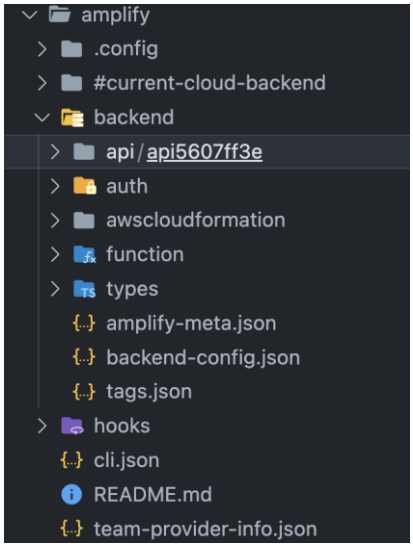


Figure 12: AWS

4.8 System Architecture

4.8.1 Database Design

The database contains 4 models, namely User, Items, Auction, and Transaction.

User Model

No.	Property	Data Type
1.	_id	Object
2.	name	String

3.	contact_number	Number
4.	email	String
5.	profile_picture	String

Items Model

No.	Property	Data Type
1.	_id	Object
2.	brand	String
3.	model	String
4.	price	Number
5.	description	String
6.	registration_date	String
7.	images	Array
8.	seller_id	String
9.	seller_name	String
10.	seller_image	String
11.	sold	Boolean
12.	cart	Array

Auction Model

No.	Property	Data Type
1.	_id	Object
2.	brand	String
3.	model	String
4.	buyout_price	Number
5.	starting_bid	Number
6.	reserve_price	Number
7.	ending_time	String
8.	description	String
9.	registration_date	String
10.	images	Array
11.	seller_id	String
12.	seller_name	String
13.	seller_image	String
14.	sold	Boolean
15.	highestBidder	String
16.	highestPrice	Number

Transaction Model

No.	Property	Data Type
1.	_id	Object
2.	item_id	String
3.	user_id	String
4.	user_name	String
5.	bid_price	String

4.8.2 MongoDB connection

The tables of the database are listed down and is connected to the local backend, Node.js, through two libraries, dotenv for keeping the connection link and port number in a hidden, protected file and mongoose for the connection between MongoDB and the Node.js files.

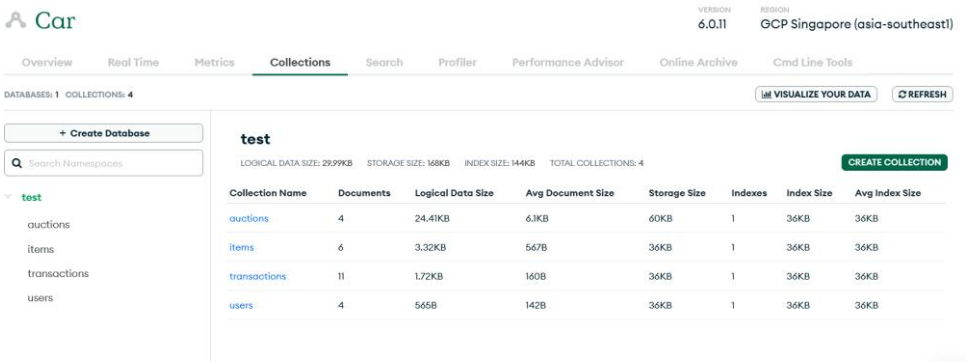


Figure 13. MongoDB database page

```
mongoose.connect(process.env.MONGO_URI)
    .then(() =>{
        app.listen(process.env.PORT, () => {
            console.log('Connected to database and port', process.env.PORT)
        })
    })
    .catch((error)=>{
        console.log(error)
    })
```

Figure 14. Node.js connection code using mongoose library (dotenv not shown to protect privacy)

4.8.3 API Design

Upon identification of the use cases and requirements and a comprehensive understanding of the database model, this API design provides endpoints for

managing cars, users, auctions, and transactions. Each endpoint serves a specific purpose and supports various operations. The following table gives an overview of the API endpoints in our design.

Below table is the list of endpoints used in our project.

No.	Endpoint	Description
1.	GET /	Retrieves a list of all cars
2.	GET /:seller/list	Retrieves a list of all cars posted by a specific seller
3.	GET /:seller/pastlist	Retrieves a list of all cars sold by a specific seller
4.	POST /createlist	Creates a new car listing
5.	PATCH /updatelist/:id	Updates a car's details by its ID
6.	GET /user/:id	Retrieves a user by their ID
7.	POST /createuser	Creates a new user
8.	GET /auctions	Retrieves a list of all auction cars
9.	GET /auctions/:id	Retrieves a single auction car by its ID
10.	POST /createauction	Creates a new auction car listing
11.	GET /auctions/:seller	Retrieves a list of all auction cars posted by a specific seller
12.	GET /pastauctions/:seller	Retrieves a list of all auction cars sold by a specific seller
13.	PATCH /updateauction/:id	Updates an auction car's details by its ID
14.	POST /createtransaction	Creates a new transaction
15.	GET /:item_id/transaction	Retrieves all bids placed for a single auction car

Table 1. List of APIs

4.8.4 Deployment

The deployment phase of our project represented a convergence of multiple AWS services and serverless architecture, ensuring that our application was not only robust and responsive but also seamlessly integrated and efficiently managed.

For the backend, we utilized AWS EC2 (Elastic Compute Cloud) to configure and manage the server runtime. EC2 instances provided us with the flexibility to select the appropriate compute resources tailored to our application's needs, which we could scale on-demand based on the incoming traffic. This adaptability was crucial in maintaining optimal performance during varying load conditions.

Serverless architecture played a transformative role in our deployment strategy. By adopting the serverless framework, we were able to deploy our backend APIs without the need to provision or manage servers. This approach significantly reduced our infrastructure overhead and streamlined our operational workflow. With serverless, we constructed autonomous functions triggered by HTTP requests, each running in

isolation to perform specific tasks. This not only improved our application's scalability but also allowed for finer cost control, as we paid only for the compute time we consumed, down to the millisecond.

With serverless architecture also help the integration of front end and back end become easier as having internet open end-point without depending on running localhost. It help the integration better.

The interconnection between our APIs and the database was meticulously handled. We employed serverless functions to interact with our MongoDB database, executing operations such as querying, inserting, and updating documents. This interaction was facilitated through a well-defined set of API endpoints, which were securely exposed for frontend consumption.

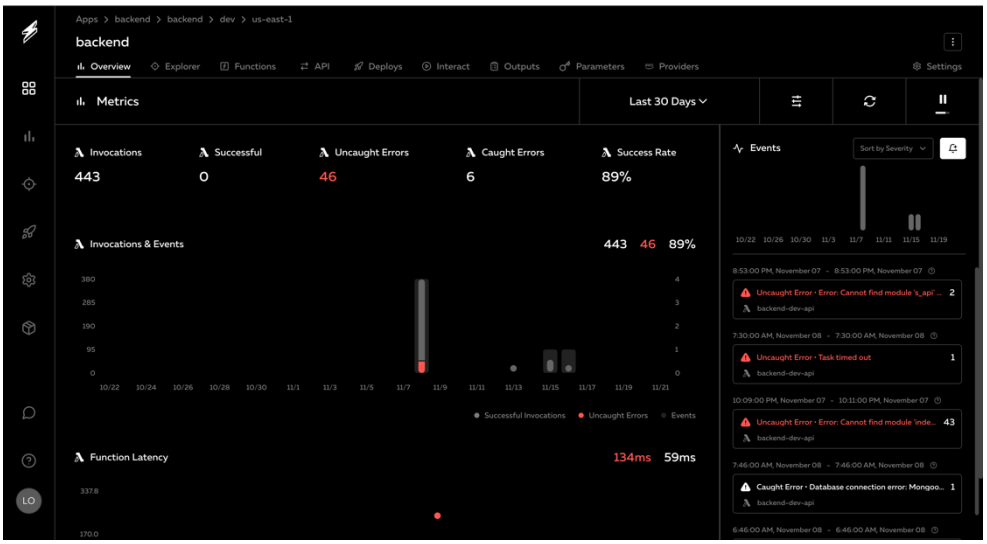


Figure 15: B.1 Serverless API

Category	Resource name	Operation	Provider plugin
Api	api5607ff3e	No Change	awscloudformation
Auth	Authentication	No Change	awscloudformation
Function	AuctionAPI	No Change	awscloudformation
Function	authentication64447f17	No Change	awscloudformation
Function	lamdaBackend	No Change	awscloudformation

Figure 16: API in Amplify

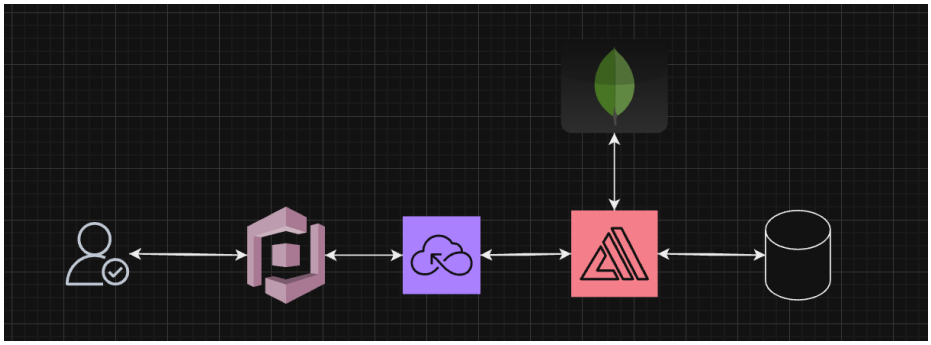


Figure 17: System Design of Flow API and Backend in Deployment

4.9 Frontend Implementation

4.9.1 Folder Structure

Our project has been divided into two major folders: frontend and assets as shown in Figure 11. The frontend folder contains JavaScript files that are used to implement our mobile application's user interface. The assets folder contains all the images as well as a fonts subfolder that contains all the font types used in our frontend JavaScript files.

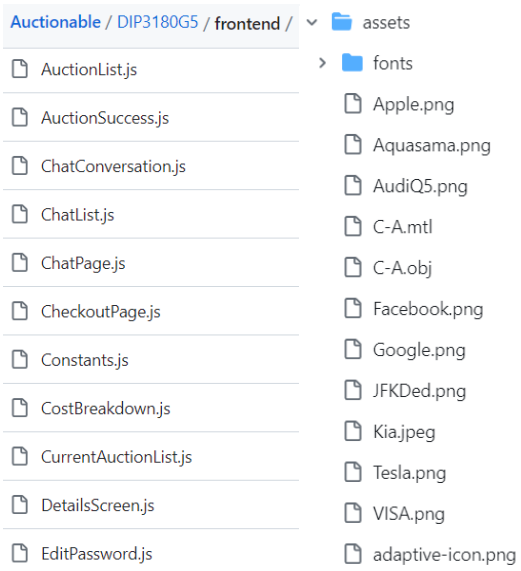


Figure 18: Folders Structure

4.9.2 Page Routing

The application will begin in the general path depicted in Figure 19 below. It will display a loading page, followed by a login page with a button that allows users to switch between this page and the registration page. If the user clicks on the forget password text, they will be taken to the forget password page.

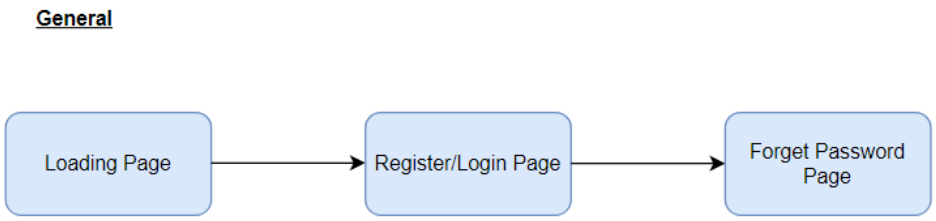


Figure 19: General Path

Upon successful login, there will be a bottom navigation bar on the screen with five separate tabs: Home, Listing, Auction, Chat, and Profile. On the following page, I will go over the page routing of each of these tabs in further detail.

The overall page routing of the Home tab is depicted in Figure 20 below. Users can access to three separate sub routes, which include looking at car listing details, checking out of standard listings, and auction listings of cars on the cart page.

Users will initially be on the home page for the first sub route **(Blue)**, which allows them to access the details of a car listing. Following that, users can either view the all-listing page by clicking the "See More" button or the Settings page if the filter button is selected. Both will take you to the details page, which provides information on the car. There will be two buttons available on the details page. One will direct users to the chat page where they can communicate with the seller, while the other will link them to the auction details page if the car is currently up for auction.

For the second sub route **(Yellow)**, where users can check out cars from the regular listing. Users can display their cart on their screen by selecting the cart icon on the home page. From here, two tabs will be displayed: one for cars added to the cart from the regular listing and another for cars added to the cart from the auction listing. If a user checkout a car from the standard listing, a pop-up will appear asking them to confirm payment, and following completion, the transaction success page will be displayed.

However, if users opt to see cars on the auction listing, which is the third sub route **(Green)**, they will be directed to the cost breakdown page, which will display all costs incurred. Selecting the instalment option on the cost breakdown page will take them to a new screen that includes the instalment option price. When consumers make a payment, regardless of whether the instalment option is selected, they will be navigated to the auction transaction success page. They can return to the cart page by clicking the "Continue" button on the page.

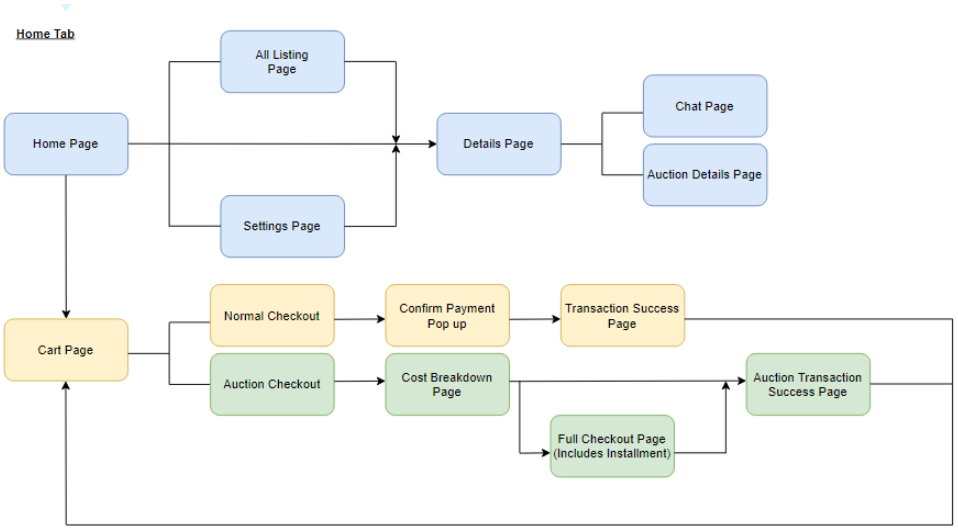


Figure 20: Home Tab Page Routing

The overall page routing of the Listing tab is shown in Figure 21 below. Users will initially be taken to the listing page, where they can view all the available listings. Users will be led to the listing category page after selecting the "+" add button on the screen, where they can choose to create a listing for normal cars or auction cars, which will navigate them to the create listing page or create auction page, respectively. If users create a listing for a regular car, they will be directed to the review listing page to confirm the details of the car before officially listing it. Both the create auction page and review listing page can navigate the user back to the listing page by pressing on a button called “Continue”.

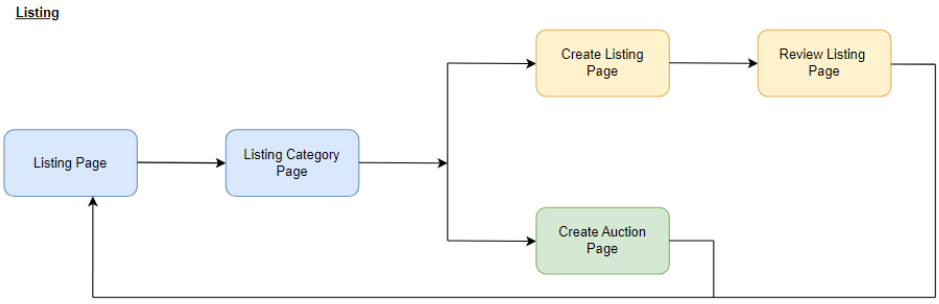


Figure 21: Listing Tab Page Routing

Figure 22 illustrates the overall page routing for the Auction tab. Users will be taken to the auction list page, where they can view all the cars that are currently up for auction. When a user clicks on a specific car on the page, they are taken to the auction details page, which displays the car image along with three separate tabs: Details, Bidding, and Leaderboard. If users want to bid, a confirmation bid pop up will appear when they click the "Submit Bid" button. The bid successful page will be displayed to users after they confirm their bid.

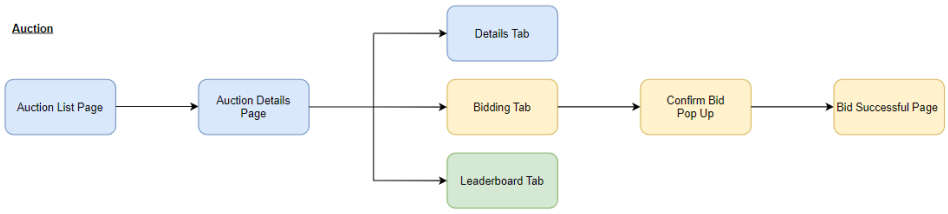


Figure 22: Auction Tab Page Routing

Figure 23 shows the overall page routing for the Chat tab. Users will begin by visiting the chat main page, which will display two different chat tabs, one with a list of chats as a seller and one with a list of chats as a buyer. Users can access individual chat conversations by selecting any individual chat featured on the chat list on either the seller or buyer tabs.

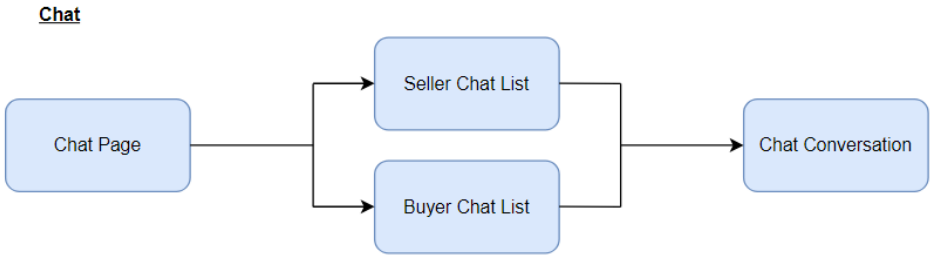


Figure 23: Chat Tab Page Routing

Figure 24 describes the overall page routing for the Profile tab. Users will first see the profile main page, which will provide a few options such as Notification, Auctions, Purchase History, and Settings. Each selection will direct users to their desired path.

Initially, users will be directed to the Notification Page where they can view all the notifications. If they select the "Bid" button on one of the notifications, they will be directed to the notification outbid page where they can click on the "Outbid Now!" button available to automatically navigate them to the bidding tab of the auction details page to submit a new bid.

Second, when users select the Auction option, they will be presented with two separate tabs: Current Auction Tab and Past Auction Tab, which will display the auctions in which they have previously and presently participated. If users click the "Outbid Now!" or "Visit Auction Now!" buttons on the current auction tab, they will be taken to the auction details page's bidding tab.

Thirdly, selecting the Purchase History option allows users to view all their previous purchases and transactions. Finally, by selecting the Settings option, users will be redirected to the edit password page, where they can change their password.

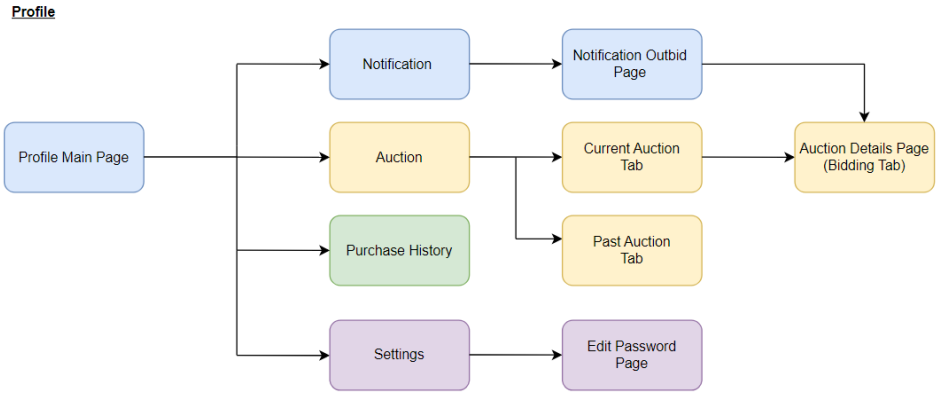


Figure 24: Profile Tab Page Routing

4.9.3 Component Development

The figure below shows some of the components obtained from React and other external libraries used in this project. These components were utilized to help develop our frontend user interfaces in line with our Figma design, making them more interactive for users. Some of these components were also reused across many JavaScript frontend files.

Components	Dependencies
Apploading	Expo-app-loading
Camera	Expo-Camera
Font	Expo-font
ImagePicker	Expo-Image-Picker
LinearGradient	Expo-Linear-Gradient
StatusBar	Expo-Status-Bar
Ionicons	Expo/Vector-Icons
React, Component	React
View, Image, Text, Touchable Opacity, StyleSheet, Modal, Flatlist, Animated, Easing, ScrollView, SafeAreaView, Pressable, AppRegistry, ImageBackground, Dimensions, TouchableWithoutFeedback, TextInput, KeyboardAvoidingView, Button, LogBox, Alert	React-Native Components
Animatable	React-Native-Animatable
RangeSlider	React-Native-Assets/Slider
ConfettiCannon	React-Native-Confetti-Cannon
CountDown	React-Native-CountDown
CheckBox	React-Native-Elements
Dropdown	React-Native-Elements-Dropdown
Bubble, GiftedChat, Send	React-Native-Gifted-Chat
MaskedView	React-Native-Masked-View/Masked-View
RadioButton	React-Native-Paper
Table, Row	React-Native-Table-Component
FontAwesome	React-Native-Vector-Icons
MaterialCommunityIcons	React-Native-Vector-Icons/AntDesign
Icon	React-Native-Vector-Icons/FontAwesome
MaterialCommunityIcons	React-Native-Vector-Icons/MaterialCommunityIcons
createBottomTabNavigator, createMaterialTopTabNavigator	React-Navigation
NavigationContainer, useIsFocused, getFocusedRouteNameFromRoute	React-Navigation/Native
createStackNavigator	React-Navigation/Stack

Figure 25: Components from React or external libraries

The frontend team also created customized components that were used across many JavaScript files, as seen in Figure 19 below. Majority of these components were primarily used for screen navigation, but some were also used to call upon their own functions, which could be used globally across other JavaScript files.

Components	
AuctionDetailsPage	NotificationMain
AuctionListPage	NotificationOutbid
AuctionSuccessPage	PastAuctionList
ChatConversation	ProfileAuctionPage
ChatList	ProfilePage
ChatPage	ProfileSettingsPage
CheckoutPage	PurchaseHistoryPage
CostBreakdownPage	RegisterPage
createListPage	SettingsPage
CurrentAuctionList	TransactionAuctionPage
DetailsPage	TransactionScreen
HomePage	useWishlist
ListingCategoryScreen	WishlistPage
ListingPage	WishlistProvider
LoadingPage	

Figure 26: Customized Components

4.9.4 Integration

The frontend and backend components are connected through asynchronous JavaScript, await and fetch. To obtain the JSON object from the backend, we call a GET HTTP request via Application Programming Interface (API), with the code as follows:

```
const fetchListingsData = async () => {
  let response, data;
  try {
    response = await fetch(
      "http://localhost:4000/api/cars"
    );
    data = await response.json();
    if (data === undefined){
      setAllListingsData(AllListingsDataConstants);
    }
    setAllListingsData(data);
  } catch (error) {
    setAllListingsData(AllListingsDataConstants);
  }
};

useEffect(() => {
  fetchListingsData();
  getFonts().then(() => setFontsLoaded(true));
}, []);
```

Figure 27: GET HTTP Request

We implement this in these pages: HomeScreen, AllListing, AuctionDetails, DetailsScreen, ListingScreen, and WishlistPage.

Besides GET request, we also implement POST HTTP request via API to create data into the database, with the code as follows:

```
  try {
    console.log(dataToSubmit);
    const response = await fetch('http://localhost:4000/api/cars/createlist', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(dataToSubmit),
    });

    if (response.ok) {
      console.log('Form data submitted successfully:', dataToSubmit);
      navigation.navigate('ListingPage', dataToSubmit);
    } else {
      console.error('Failed to submit form data:', response.statusText);
    }
  } catch (error) {
    console.error('Error submitting form data:', error.message);
  }
};
```

Figure 28: POST HTTP Request

We implement this in these pages: CreateListingScreen, CreateAuctionScreen.

We also implement PATCH HTTP request via API to update the data in the database, with the code as follows:

```
try {
  const response = await fetch(`http://localhost:4000/api/cars/updateauction/id=${itemId}`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(updatedItemData),
  });

  if (response.ok) {
    console.log('Item added to cart successfully:', updatedItemData);
  } else {
    console.error('Failed to add item to cart:', response.statusText);
  }
} catch (error) {
  console.error('Error adding item to cart:', error.message);
}
};
```

Figure 29: PATCH HTTP Request

We implement this in these pages: ItemsUpdate, AuctionDetails, DetailsScreen.

5. Conclusion and Recommendation

5.1 Conclusion

After many trials and tribulation, with countless of hours testing, fixing, and debugging code, we have successfully achieved our given goal and vision. With only 14 weeks with modules, we need to study and starting our project with little to know knowledge of the programming knowledge it requires us, we managed to create a complete product. While there are some parts which can be further improved on such as design consistency, it is still a project we are proud with given by the restrictions and limitation given to us.

After this project, we have left with more knowledge of mobile app development and working in a team. We learn about working as a team, having an effective communication and weekly team updates help with our progress. Understanding how a team functions help with future endeavours. We resolve each other's doubts and help with any problems that may arose from our teammates. We are grateful for each other and hoped we have another opportunity to work together as a team.

5.2 Recommendation for Future Works

1. Use data analytics to provide accurate insights on the true value of the car to sellers and buyers, based on previous transactions.

2. Introduce cybersecurity protocols against potential DDOS attacks that may disrupt the real-time auction.

3. Implement AI and machine learning to recommend vehicles suitable to the customer's needs and budget.

References

[1] Z. Lee, “Singapore’s Future Decacorn? Carro Has Big Ambitions For The Online Car Market In Southeast Asia,” *Forbes*. 12 Aug 2021 [Online]. Available: https://www.forbes.com/sites/zinnialee/2021/08/12/singapores-future-decacorn-carro-has-big-ambitions-for-the-online-car-market-in-southeast-asia.

[2] Carro, “About Us. Challenging the status quo in the automotive industry,” *Carro*. 2023. Available: https://carro.sg/about.

Appendices:

A. (For project with hardware expenses) Bill of Materials (BOM) – “All spending” AND “for the project prototype”
B. Design Diagrams (e.g., detailed circuit diagrams, software engineering diagrams)
C. User Guide
D. Maintenance Guide (How to resolve issue)
E. How-To Guides
F. Source Code
G. Others

Source Code

Server.js

```
require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
let server = {};

const startServer = (port) => {
    if (port === undefined)
        port = process.env.PORT;

    const app = express();
    app.use(express.json());

    app.use((req, res, next) => {
        console.log(req.path, req.method);
        next();
    });

    mongoose.connect(process.env.MONGO_URI)
        .then(() => {
            const carRoutes = require('./routes/lists');
            app.use('/api/cars', carRoutes);
            console.log("connecting to port " + port);
            server[port] = app.listen(port, () => {
                console.log('Connected to the database and listening on port', port);
            });
            console.log(server[port]);
        })
        .catch((error) => {
            console.log(error);
        });
};
```

```
function stopServer(port) {
    if (port === undefined)
    port = process.env.PORT;
    console.log("closing port " + port);

    if (server[port] && server[port].listening) { // Check if server is running
        server[port].close((err) => {
            if (err) {
                console.error('Error closing the server:', err);
            }
        });
    } else {
        console.error('Server is not running. Cannot close.');
    }
}

if (require.main === module) {
    startServer();
}

module.exports = { startServer, stopServer };
```

Lists.js (Routing System)

```
const express = require('express')
const router = express.Router();
// import your car controller
const carController = require('../controllers/carController')
const userController = require('../controllers/userController')
const auctionCarController = require('../controllers/auctionController')
const transactionController = require('../controllers/transactionController')

// car controller
router.get('/', carController.getAllCars);
router.get('/id=:id', carController.getCarDetails);
router.get('/:seller/list', carController.getItems)
router.get('/:seller/pastlist', carController.getSoldItems)
router.get('/:buyer/cart', carController.getCartItems)
router.post('/createlist', carController.createItem)
router.patch('/updatelist/id=:id', carController.updateItem)

// user controller
router.get('/user/id=:id', userController.getUser)
router.post('/createuser', userController.createUser)

// auction controller
router.get('/auctions', auctionCarController.getAllAuctionCars)
router.get('/auctions/id=:id', auctionCarController.getAuction) // get a single auction car
router.post('/createauction', auctionCarController.createAuction)
router.get('/auctions/seller=:seller', auctionCarController.getAuctionItems) // get a list of auction cars posted by a specific seller
router.get('/pastauctions/seller=:seller', auctionCarController.getAuctionSoldItems) // get a list of sold auctions cars by a specific seller
router.patch('/updateauction/id=:id', auctionCarController.updateAuctionItem) // this is to update the current highest bidder
router.get('/:buyer/auctionCart', auctionCarController.getAuctionCart)

// transaction controller
router.post('/createtransaction', transactionController.createTransaction) // changed
router.get('/:item_id/transaction', transactionController.getTransaction) // changed, everyone's transactions for a specific auction car (use in leaderboard)

module.exports = router;
```

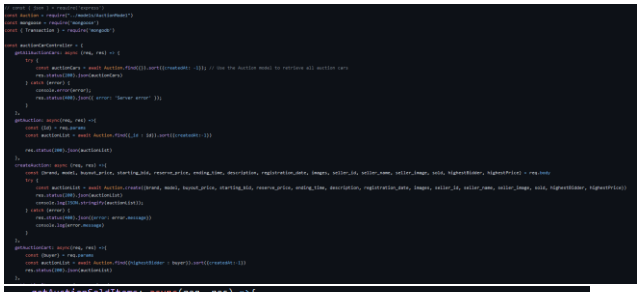
Controllers:

ItemsController

UserController

```
const User = require('../models/UserModel')
const mongoose = require('mongoose')

const userController = {
    getUser: async(req, res) => {
        const {id} = req.params

        if (!mongoose.Types.ObjectId.isValid(id)) {
            return res.status(404).json({error: 'No User'})
        }

        const user = await User.findById(id)

        if(!user) {
            return res.status(404).json({error: 'No User'})
        }

        res.status(200).json(user)

    },
    createUser: async (req, res) =>{
        const {name, contact_number, email} = req.body

        try {
            const user = await User.create({name, contact_number, email})
            res.status(200).json(user)
        } catch (error) {
            res.status(400).json({error: error.message})
        }
    }
};

module.exports = userController;
```

AuctionController



```
    getAuctionSoldItems: async(req, res) =>{
        const {seller} = req.params
        const soldItems = await Auction.find({seller: seller, sold: true}).sort({createdAt:-1})

        if (!soldItems) {
            return res.status(400).json({error: 'No previous items'})
        }
        res.status(200).json(soldItems)
    },
    updateAuctionItem: async(req, res) =>{
        const {id} = req.params

        if (!mongoose.Types.ObjectId.isValid(id)) {
            return res.status(404).json({error: 'No such id'})
        }
        const auctionList = await Auction.findOneAndUpdate({_id: id},{
            ...req.body
        })
        if (!auctionList) {
            return res.status(400).json({error: 'No previous items'})
        }
        res.status(200).json(auctionList)
    }
}

module.exports = auctionCarController;
```

TransactionController

```
const Transaction = require('../models/TransactionModel')
const mongoose = require('mongoose')

const transactionController = {
    getTransaction: async(req, res) =>{
        const {item_id} = req.params
        const transactionList = await Transaction.find({item_id : item_id}).sort({createdAt:-1})

        res.status(200).json(transactionList)
    },
    createTransaction: async (req, res) =>{
        const {item_id, user_id, user_name, bid_price} = req.body

        try {
            const transactionList = await Transaction.create({item_id, user_id, user_name, bid_price})
            res.status(200).json(transactionList)
            console.log(JSON.stringify(transactionList));
        } catch (error) {
            res.status(400).json({error: error.message})
            console.log(error.message)
        }
    }
}

module.exports = transactionController
```

Models:

CarModel

```
const mongoose = require('mongoose')

const Schema= mongoose.Schema

const itemSchema = new Schema({
    brand:{
        type: String,
        required: true
    },
    model:{
        type: String,
        required: true
    },
    price:{
        type:Number,
        required: true
    },
    description: {
        type: String,
        required: true
    },
    registration_date:{
        type: String,
        required: true
    },
    images:{
        type: [String],
        required: true
    },
    seller_id:{
        type: String,
        required: true
    },
    seller_name:{
        type: String,
        required: true
    },
    seller_image:{
        type: String,
        required: true
    },
    sold:{
        type: Boolean,
        required: true
    },
    cart:{
        type: [String],
        required: true
    },
}, { timestamps: true })

module.exports = mongoose.model('Items', itemSchema)
```

AuctionModel

```
const mongoose = require('mongoose')

const Schema= mongoose.Schema

const auctionSchema = new Schema({
    brand:{
        type: String,
        required: true
    },
    model:{
        type: String,
        required: true
    },
    buyout_price:{
        type:Number,
        required: true
    },
    starting_bid:{
        type: Number,
        required: true
    },
    reserve_price: {
        type: Number,
        required: true
    },
    ending_time:{
        type:String,
        required: true
    },
    description: {
        type: String,
        required: true
    },
    registration_date:{
        type:String,
        required: true
    },
    images:{
        type: [String],
        required: true
    },
    seller_id:{
        type: String,
        required: true
    },
    seller_name:{
        type: String,
        required: true
    },
    seller_image:{
        type: String,
        required: true
    },
    sold:{
        type: Boolean,
        required: true
    },
    highestBidder: {
        type: String, // id of highest bidder
        required: true
    },
    highestPrice: {
        type: Number,
        required: true
    },
}, { timestamps: true })

module.exports = mongoose.model('Auction', auctionSchema)
```

UserModel

```
const mongoose = require('mongoose')

const Schema= mongoose.Schema

const userSchema = new Schema({
    name: {
        type: String,
        required: true
    },

    contact_number: {
        type: Number,
        required: true
    },

    email: {
        type: String,
        required: true
    },

    profile_picture: {
        type: String,
        default: 'NA',
        required: true
    }
}, { timestamps: true })

module.exports = mongoose.model('User', userSchema)
```

TransactionModel

```
const mongoose = require('mongoose')

const Schema= mongoose.Schema

const transactionSchema = new Schema({
    item_id: {
        type: String,
        required: true
    },

    user_id: {
        type: String,
        required: true
    },

    user_name: {
        type: String,
        required: true
    },

    bid_price: {
        type: Number,
        required: true
    },
}, { timestamps: true })

module.exports = mongoose.model('Transaction', transactionSchema)
```